

Reducing Procedural Code

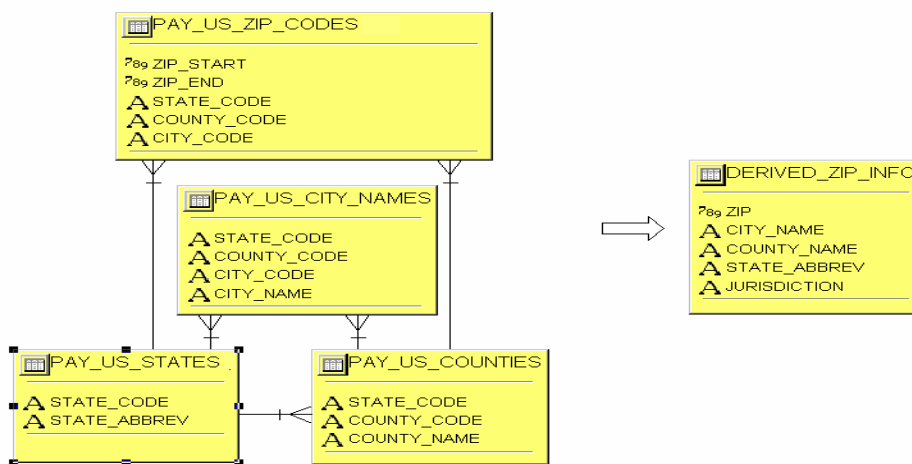
Parin Jhaveri and Lev Molyaner
 1 January, 2004

By taking full advantage of the power of SQL, programmers can improve performance and simplify code. In this article, Parin Jhaveri and Lev Molyaner examine two common business problems, present choices of solutions, and draw conclusions after comparing results.

Problem 1: Definition

During development of a web-enabled extension for Oracle Payroll for a client, we faced an interesting performance problem. We needed to derive a de-normalized table from four different tables as shown in Figure 1.

Figure 1. Problem Definition



The main source table, PAY_US_ZIP_CODES table contained one row for each range of zip codes defined by zip_start and zip_end columns. The target table DERIVED_ZIP_INFO required one row for each zip as shown in Figure 2. The highlighted rows in Figure 2 indicate that the target table will often contain more than one row for each row in PAY_US_ZIP_CODES table.

Figure 2. Sample Data

PAY_US_ZIP_CODES					DERIVED_ZIP_INFO				
ZIP_START	ZIP_END	ST	COU	CITY	ZIP	CITY_NAME	COUNTY_NAME	ST	JURISDICTION
00601	00601	72	001	0010	00601	Adiuntas	Puerto Rico County L	PR	72-001-0010
00602	00602	72	001	0020	00602	Aguada	Puerto Rico County L	PR	72-001-0020
00602	00602	72	001	0030	00602	Aguadilla	Puerto Rico County L	PR	72-001-0030
00603	00605	72	001	0030	00603	Aguadilla	Puerto Rico County L	PR	72-001-0030
					00603	Victoria Street	Puerto Rico County L	PR	72-001-0030
					00604	Aguadilla	Puerto Rico County L	PR	72-001-0030
					00604	Barrio Escobar	Puerto Rico County L	PR	72-001-0030
					00605	Aguadilla	Puerto Rico County L	PR	72-001-0030
					00605	Barrio Escobar	Puerto Rico County L	PR	72-001-0030

Solution Overview

There are several different ways to address this problem. We will examine following three solutions and compare their performance:

- PL/SQL stored procedure
- PL/SQL stored procedure with Bulk Inserts
- SQL Statement

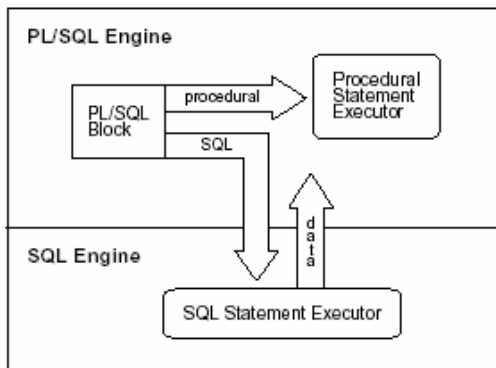
Solution I – PL/SQL Stored procedure

The first solution that comes to mind is to use a PL/SQL stored procedure, which iterates through the cursor and inserts one row for each zip into the target table (See Listing 1). Since the PAY_US_ZIP_CODES table contains one row per range of zip codes, we need to generate one row for each zip code between the starting and the ending zip. This is achieved by using a LOOP. When this procedure is executed, the PL/SQL engine executes all the procedural statements but dispatches each SQL statement to the SQL Engine. This switch from the PL/SQL engine to the SQL engine is called a context switch (see Figure 3). Each context switch adds to overhead of our program. Since our procedure calls the INSERT statement within the FOR-LOOP, it performs one context switch for each iteration through the loop. As a result of these excessive context switches, performance suffers.

Listing 1 PL/SQL Stored Procedure

```
declare
  cursor c_zip
  is
    select state.state_abbrev
           ,county.county_name
           ,city.city_name
           ,zip.zip_start
           ,zip.zip_end
           ,zip.state_code || '-'
             || zip.county_code || '-'
             || zip.city_code jurisdiction
    from   pay_us_city_names      city
           ,pay_us_zip_codes      zip
           ,pay_us_states         state
           ,pay_us_counties       county
    where  zip.state_code        = city.state_code
    and    zip.county_code       = city.county_code
    and    zip.city_code        = city.city_code
    and    county.state_code     = city.state_code
    and    county.county_code    = city.county_code
    and    state.state_code      = city.state_code;
begin
  for r_zip in c_zip loop
    for ln_zip_code in to_number(r_zip.zip_start)
      ..to_number(r_zip.zip_end) loop
      insert into derived_zip_info
        values (lpad(ln_zip_code,5,'0')
               ,r_zip.city_name
               ,r_zip.county_name
               ,r_zip.state_abbrev
               ,r_zip.jurisdiction);
    end loop;
    commit;
  end loop;
end;
```

Figure 3. Context Switches



Solution II - PL/SQL Stored procedure with Bulk Inserts

One way to reduce context switches is to use a bulk insert. We can enhance our previous solution as shown in Listing 2. The FOR-CURSOR loop now populates PL/SQL collections. After the loop, our process uses a FORALL bulk insert to insert all rows using a single SQL statement. Because of reduced context switches, we expect performance of this solution to be significantly better than our first solution.

Listing 2. PL/SQL Procedure with Bulk Inserts

```

...
begin
    ln_ctr:=0;
    for r_zip in c_zip loop
        for ln_zip_code in to_number(r_zip.zip_start)
            ..to_number(r_zip.zip_end) loop
                ln_ctr := ln_ctr+1;
                l_ziptab(ln_ctr) := lpad(ln_zip_code,5,'0');
                l_citytab(ln_ctr) := r_zip.city_name;
                l_countytab(ln_ctr) := r_zip.county_name;
                l_statetab(ln_ctr) := r_zip.state_abbrev;
                l_juristab(ln_ctr) := r_zip.jurisdiction;
            end loop;
        end loop;
    forall ln_i in 1..ln_ctr
        insert into derived_zip_info
        values (l_ziptab(ln_i)
            ,l_citytab(ln_i)
            ,l_countytab(ln_i)
            ,l_statetab(ln_i)
            ,l_juristab(ln_i));
    end;
...

```

Our current solution uses one collection for each column in the target table. We could not use a single collection for multiple columns in a bulk operation because it was not supported prior to Oracle 9i Release 2. In Release 2, we can create one PL/SQL collection based on the target table and use it in the bulk insert (see Listing 3).

Listing 3. PL/SQL Procedure with Bulk Inserts (Oracle 9i2)

```
...
type zip_tab is table of derived_zip_info%ROWTYPE
  index by pls_integer;
lt_z zip_tab;
begin
  ln_ctr:=0;
  for r_zip in c_zip loop
    for ln_zip_code in to_number(r_zip.zip_start)
      ..to_number(r_zip.zip_end) loop
      ln_ctr := ln_ctr+1;
      lt_z(ln_ctr).zip      := lpad(ln_zip_code,5,'0');
      lt_z(ln_ctr).city_name := r_zip.city_name;
      lt_z(ln_ctr).county_name := r_zip.county_name;
      lt_z(ln_ctr).state_abbrev:= r_zip.state_abbrev;
      lt_z(ln_ctr).jurisdiction:= r_zip.jurisdiction;
    end loop;
  end loop;
  forall ln_i in 1..ln_ctr
    insert into derived_zip_info
      values lt_z(ln_i);
  end;
...

```

Solution III - SQL Statement

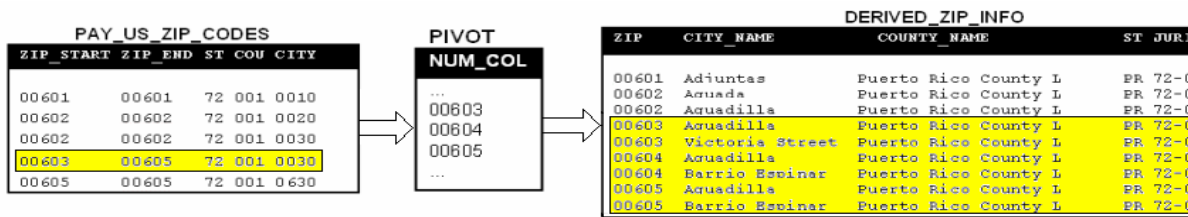
We can improve performance further by completely eliminating the FOR-CURSOR loop. Listing 4 shows a solution, which uses a single SQL statement. This approach can be accomplished in a single SQL statement by utilizing a pivot table. A pivot table allows us to utilize a technique of joining to a dummy table to produce multiple rows for each source row. In our case, this table contains a single column to cover a complete range of zips (i.e. 1 to 99999). We join to this table to produce one row for each zip in the range of zips defined by our start zip and end zip columns (see Figure 4). To avoid a full table scan of the pivot table, we have created an index on num_col - the only column in pivot table. Another performance improvement is to create the pivot table as an index-organized table (IOT). This solution performs much better than the first two because it completely eliminates the procedural looping required in the first two solutions.

Listing 4 SQL Solution

```
insert into derived_zip_info
  (zip
  ,city_name
  ,county_name
  ,state_abbrev
  ,jurisdiction)
(select lpad(pivot.num_col,5,'0')
  ,city.city_name
  ,county.county_name
  ,state.state_abbrev
  ,zip.state_code || '-'
  || zip.county_code || '-'
  || zip.city_code jurisdiction
from   pay_us_city_names    city
      ,pay_us_zip_codes     zip
      ,pay_us_counties      county
      ,pay_us_states        state
      ,pivot
where  zip.state_code       = city.state_code
and    zip.county_code      = city.county_code
and    zip.city_code        = city.city_code
and    county.state_code    = city.state_code
and    county.county_code   = city.county_code
and    state.state_code     = city.state_code
and    pivot.num_col
  between zip.zip_start and zip.zip_end);

```

Figure 4. Using Pivot Table



In Oracle 9i, you can avoid creating a physical pivot table by defining and using a parallel pipe-lined table function to generate numbers from 1 and 99999. Listing 5 shows such pipe-lined table function.

Listing 5 Using Pivot Function in Oracle 9i

```

create or replace type pivot_rec as object
  (num_col number(6));

create or replace type pivot_tab as table
  of pivot_rec;

create or replace package pivot as
  function generate_number (max_num in number)
    return pivot_tab parallel_enable pipelined;
end;

create or replace package body pivot as
  function generate_number (max_num in number)
    return pivot_tab parallel_enable pipelined is
    outrow pivot_rec := pivot_rec(0);
  begin
    for ln_i in 0..max_num loop
      outrow.num_col := ln_i;
      pipe row(outrow);
    end loop;
    return;
  end generate_number;
end;

```

Our solution can now be enhanced to physical pivot table with the new pivot table function as follows:

```

...
from   pay_us_city_names      city
       ,pay_us_zip_codes      zip
       ,pay_us_counties       county
       ,pay_us_states         state
       ,(select num_col from
          table(pivot.generate_number(99999))) pivot
where  zip.state_code        = city.state_code
...

```

Comparison of all three solutions

In summary, the first solution has the overhead of excessive context switches. The second solution reduces context switches but still requires a loop to populate a PL/SQL collection. The third solution completely eliminates the looping. In order to compare these three solutions, we performed benchmark. Our benchmark is based on generating approximately 500,000 rows in the target table. For consistency of this benchmark, we executed all three solutions repeatedly on the same database server. Table 1 outlines the results of the benchmark. As expected, each solution performance significantly better than the prior one!

Table 1 Comparison Run Times

Method	Run Time (in minutes and seconds)
PL/SQL Stored Procedure	4:10
PL/SQL Stored Procedure with Bulk Inserts	1:50
SQL Statement	0:40

Problem 2: Definition

Another common business problem is to combine related history from multiple tables into one table. For our example, we will use three source tables, which contain information about a person, called t1, t2, and t3. Each table contains person id (pid), start date, and end date columns. The purpose of each table is to store some information about a person that can change independently of the other information. Thus, each table has one or more mutually exclusive columns with information about the person. The primary key of each table is person id and start date. Our goal is to create one table, which combines information from all three tables. The primary key of this table is also person id and start date but this table has one record for every change in any of the three source tables. See Figure 5 for sample data.

Figure 5 Sample Data

T1				
PID	SD	ED	X	CBY
1	28-Sep-01	8-Oct-01	12	Ann
1	9-Oct-01	23-Jun-04	20	Joe
2	28-Sep-01	3-Oct-01	12	Ann
2	4-Oct-01	23-Jun-04	20	Joe

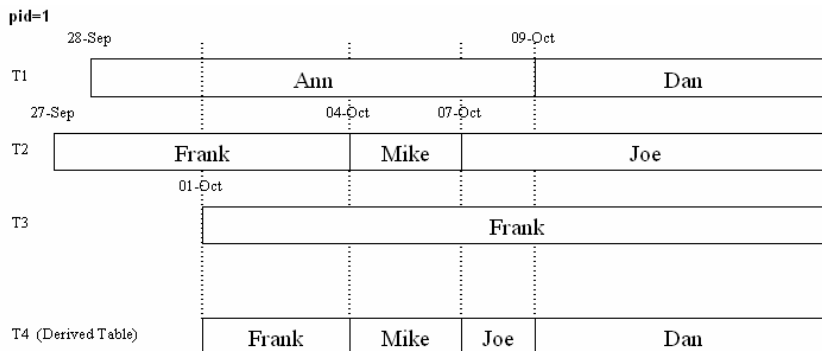
T2				
PID	SD	ED	Y	CBY
1	27-Sep-01	3-Oct-01	a	Frank
1	4-Oct-01	6-Oct-01	b	Mike
1	7-Oct-01	23-Jun-04	c	Joe
2	28-Sep-01	01-Oct-01	a	Frank
2	4-Oct-01	6-Oct-01	b	Mike
2	7-Oct-01	23-Jun-04	c	Joe

T3				
PID	SD	ED	Z	CBY
1	01-Oct-01	23-Jun-04	10	Frank
2	28-Sep-01	3-Oct-01	10	Lucy
2	4-Oct-01	23-Jun-04	30	Dan

D							
PID	SD	ED	X	Y	Z	CBY	
1	1-Oct-01	3-Oct-01	12	a	10		Frank
1	4-Oct-01	6-Oct-01	12	b	10		Mike
1	7-Oct-01	8-Oct-01	12	c	10		Joe
1	9-Oct-01	23-Jun-04	20	c	10		Joe
2	28-Sep-01	3-Oct-01	12	a	10		Ann
2	4-Oct-01	6-Oct-01	20	b	30		Joe
2	7-Oct-01	23-Jun-04	20	c	30		Joe

The start date is entered by the user, while the end date is derived by a trigger as start date of next record - 1 or some hard coded date in the future (we will use Jun 23, 2004). This means that in each table, the timeline is continuous and there are no gaps or overlaps in dates. However, the earliest record for each person will vary. See Figure 6 for a sample timeline.

Figure 6. Timeline for pid=1



In summary:

- There should be one record for each period created by a change in any of the three tables
- The table should only contain periods for which there is data in all three tables. This means that if one source table has no records for a person, the person will not appear in the target table.

Solution I - Procedural Solution

Once again the easiest solution is a “brute force” method. The brute force method uses a cursor to select from t1 table and queries the other two tables, t2 and t3, inside the t1 cursor loop. The two queries will use the current start date from t1 table in the where clause. This solution is easy to understand, but it performs poorly because of the context switches we discussed in the previous example.

Solution II – Procedural Solution

We can improve on the first solution with an algorithm shown in Listing 6. This algorithm performs a sort-merge join of 3 tables using PL/SQL. This is accomplished by defining a cursor for each table and ordering each cursor by person id and start date. It then synchronizes the rows from three cursors by using least and greatest functions on dates to control which cursors to fetch next. This solution is easy to understand and is relatively efficient because it contains only three full table scans. Alternatively, you can use UNION ALL of three tables so that the data is already sorted by person id and start date across all three tables. This is slightly simpler solution and will yield comparable performance to the above. Once again by using Bulk Inserts, you can improve the performance even further in all PL/SQL stored procedures.

This algorithm reduces the number of SQL statements significantly by taking them out of the loop because each table is accessed once in a cursor. This approach will be much more efficient because it reduces IO and the number of context switches.

Listing 6. Sort-Merge Join Solution

```
...
cursor c1 is select * from t1 order by pid, sd;
cursor c2 is select * from t2 order by pid, sd;
cursor c3 is select * from t3 order by pid, sd;
begin
...
fetch c1 into r1;
fetch c2 into r2;
fetch c3 into r3;
while c1%found and c2%found and c3%found
loop
  if r1.pid = r2.pid and r1.pid = r3.pid then
    lv_lst_ed := least(r1.ed,r2.ed,r3.ed);
    lv_grt_sd := greatest(r1.sd,r2.sd,r3.sd);
    ...
    if lv_lst_ed >= lv_grt_sd then
      insert into t4 ...
    end if;

    if r1.ed = lv_lst_ed then
      fetch c1 into r1;
    elsif r2.ed = lv_lst_ed then
      fetch c2 into r2;
    else
      fetch c3 into r3;
    end if;
  elsif r1.pid <= r2.pid and r1.pid <= r3.pid then
    fetch c1 into r1;
  elsif r2.pid <= r1.pid and r2.pid <= r3.pid then
    fetch c2 into r2;
  else
    fetch c3 into r3;
  end if;
end loop;
end;
```

Solution III - SQL Alternative

The final solution is to solve the entire problem with a single SQL Statement (see Listing 7). This solution demands some thinking about the date joins in the WHERE clause. Basically, it is retrieving all records when the greatest of the start date in all three tables is less than or equal to the least of the end date in all three tables for each person. This solution eliminates PL/SQL code completely and performs much better than the best PL/SQL solution.

Listing 7. SQL Solution

```
insert into t4
select t1.pid person_id
      ,greatest(t1.sd,t2.sd,t3.sd) start_date
      ,least(t1.ed,t2.ed,t3.ed) end_date
      ,t1.x
      ,t2.y
      ,t3.z
      ,decode(greatest(t1.sd,t2.sd,t3.sd),t1.sd,t1.cby
              ,t2.sd,t2.cby
              ,t3.sd,t3.cby) changed_by
from t1, t2, t3
where t2.pid = t1.pid
and t3.pid = t1.pid
and t1.sd <= t2.ed
and t1.ed >= t2.sd
and t1.sd <= t3.ed
and t1.ed >= t3.sd
and t2.sd <= t3.ed
and t2.ed >= t3.sd
```

Comparison

Once again, in order to compare our three solutions, we performed a benchmark. In our benchmark we had 25000 rows for t1, 10000 rows for t2 and 50000 rows in t3. Once again, to perform clean benchmark, we ran all solutions repeatedly on the same database server. Table 2 shows the results.

Table 2 Comparison Run Times

Method	Run Time (in minutes and seconds)
Brute Force PL/SQL Stored Procedure	65:00
More efficient PL/SQL Stored Procedure	6:00
SQL Statement	1:30

Conclusion

Two examples discussed in this article highlight an important point: Most SQL developers think procedurally. This means we write procedural programs, which read or write data as required. This results in programs which contain simple SQL statements nested within procedure calls or loops. The alternative is to be more SQL oriented. Design algorithms to minimize the number of SQL statements within loops. This means fewer complex SQL statements rather than more simple SQL statements. Such algorithms may seem more difficult to develop and maintain. However, once a developer shifts the mindset to SQL and data sets, the algorithms are actually easier to develop. In fact, our experience has proven that SQL oriented algorithms are more structured and result in less code. Thus maintenance is actually reduced. One recommendation is to have lots of meaningful comments within the SQL statements themselves. Most importantly SQL oriented algorithms are more efficient than their procedural counter parts most of the time!